Machine Learning at NAS

Machine learning, deep learning, and artificial intelligence have become essential tools for handling and gaining insight from the enormous amounts of data that are being generated via high-performance computing, modern modeling and simulation, and instrument technology.

These methods organize data in a way that is both *systematic* (collected, processed, and stored methodically according to a standard practice) and *semantic* (unambiguous and logical, in both order and representation, so that relationships and biases can be easily explored). Once the data is organized, it can then be analyzed using powerful statistical methods.

HECC Data Science Team: Pilot Projects

HECC offers a range of services for researchers moving into advanced data science, using the latest technologies in statistics-based data analytics, machine learning, and deep learning. For general information, see HECC Data Science Services.

The Data Science Team is currently working with NASA researchers on a number of projects, including:

- Carbon Nanotube Gas Sensor Using Neural Networks (PDF)
- Predicting Composition of Photo Voltaic Cells Using Neural Networks (PDF)

For more information about working with our team, please contact us at dataanalytics@nas.nasa.gov.

Getting Started with Machine Learning

In this section of the Knowledge Base, we provide information that can help you get started with a machine learning environment and short tutorials describing how to use the machine learning tools available on NAS systems. Please check back often, as new articles will become available soon.

Once you have a NAS account, you will have access to software and infrastructure for your machine learning project. If you are a new user, be sure to read through our New User Orientation and complete the first-time login process.

Conda Environments

We provide multiple conda environments that include basic machine learning packages, as well as common image processing and natural language processing packages, for your machine learning projects.

The following table shows currently available conda environments (all include GPU support):

Environment Name	Packages Included	
horovod	jupyterlab mpi4py pandas	pillow horovod tensorflow 2.8.0
mayavi	mayavi 4.8.1 jupyterlab matplotlib	mpi4py pandas scikit-learn
pyt1_13	dask jupyterlab matplotlib mpi4py pandas	pillow pytorch 1.13.1 pytorch lightning 1.8.1 scikit-learn
pyt2_0	dask jupyterlab matplotlib mpi4py pandas	pillow pytorch 2.0.0 pytorch lightning 2.0.0 scikit-learn
tf2_9	dask pandas jupyterlab matplotlib nltk	pillow scikit-learn sympy tensorflow 2.9 tf_agents

tf2_10	dask pandas jupyterlab matplotlib nltk	pillow scikit-learn sympy tensorflow 2.10 tf_agents
tf2_12	dask pandas jupyterlab matplotlib nltk	pillow scikit-learn sympy tensorflow 2.12 tf_agents
r3_6	r-base 3.6 jupyterlab r-caret r-dplyr r-essentials r-ggplot2 r-knitr r-markdown r-plyr	r-randomforest r-reticulate r-shiny r-tidyr r-tidyverse r-timedate r-xts r-zoo
r4_2	r-base 4.2 jupyterlab r-caret r-dplyr r-essentials r-ggplot2 r-knitr r-markdown r-plyr	r-randomforest r-reticulate r-shiny r-tidyr r-tidyverse r-timedate r-xts r-zoo
dask_mpi	dask dask_mpi pandas jupyterlab	matplotlib pillow scikit-learn
jupyterlab	jupyterlab	

The environments contain many of the same software programs, but the package versions may be different. Also, the environments may contain additional packages to the ones listed here. To learn more, see Using Conda Environments for Machine Learning.

The conda environments are provided via software modules in HECC system directories. For general information about NAS-provided software modules, see Software Directories and Using Software Modules.

Note: Rendering graphics using OpenAI Gym is currently unavailable.

New Packages

Miniconda has been upgraded to version 4.12.0 and is now licensed from Anaconda. The new license directly impacts how environments can be cloned. Some of the previous cloning options cannot be used with the licensed Anaconda environments. For the new cloning instructions, see Managing and Installing Python Packages for Machine Learning.

Singularity Images

We also provide a few pre-built Singularity images containing popular machine learning packages, built for use on the GPU nodes. The images are installed in the /swbuild/analytix/singularity/images directory. You can use the images as provided, or you can copy them to your /nobackup directory to modify them. To learn how to use these images, see Running a Singularity Container Image on Pleiades.

The following table shows currently available Singularity images:

Image File	Packages Included
tf2_4.sif	python 3.8 tensorflow 2.4 jupyterlab nltk

tf1_15.sif	python 3.8 tensorflow 1.15 jupyterlab matplotlib nltk numba pandas pillow scikit-learn
pyt1_8.sif	python 3.8 pytorch 1.8 jupyterlab matplotlib nltk numba pillow scikit-learn

Working with PBS

All NAS systems utilize the Portable Batch System (PBS) for batch job submission, job monitoring, and job management. To request examples of PBS scripts for Apache Spark and TensorFlow, please contact us at dataanalytics@nas.nasa.gov.

System Information

- Systems and Filesystems
- GPU nodes
- Current system status

Note: When you publish your results, please acknowledge the use of NAS resources.

Additional Resources

- HECC Webinars, including:
  - 2021 NAS Data Science Webinar
  - HECC Data Science Platform
- Machine Learning (Wikipedia)
- Machine Learning for Humans

Using Conda Environments for Machine Learning

We provide multiple conda environments for you to use for your machine learning projects. The environments are available via the conda module in the /swbuild/analytix/tools/modulefiles directory.

All of the available environments include the basic machine learning packages, as well as common image processing and natural language processing packages. You can activate the environments via an interactive session or with a PBS batch job.

For a list of environments we currently provide, along with the packages they include, see Machine Learning: Overview.

Loading the Module and Activating the Environments in Interactive Mode

First, access and load the miniconda module, which provides access to the environment:

```
% module use -a /swbuild/analytix/tools/modulefiles
% module load miniconda3/v4
```

Then, activate one of the environments (where *env_name* is the name of the environment):

- For bash: % source activate *env_name*
- For csh: % source /swbuild/analytix/tools/miniconda3_220407/bin/activate.csh *env_name*

Useful Conda Commands

After loading the module, you will have access to conda commands, including:

% conda info --envs
: Shows available environments.

% conda list -n *env_name*
: Shows installed packages within an environment.

(*env_name*)% conda deactivate
: Deactivates an environment after loading.

For more detailed documentation, see the Conda website.

Using a PBS Batch Job to Activate an Environment

You can activate your machine learning environment, run your program, and deactivate the environment in a PBS script. For example:

```
#!/bin/bash -x
#PBS -l select=1:model=sky_gpu:mpiprocs=1:ncpus=36:ngpus=4:mem=300g
#PBS -l place=scatter:excl
#PBS -q v100
#PBS -l walltime=1:00:00
#PBS -N test
#PBS -j oe

cd $PBS_O_WORKDIR

module purge

# load the module and environment
module -a use /swbuild/analytix/tools/modulefiles
module load miniconda3/v4
source activate env_name

# run python script
python test_cnn.py

# deactivate environment
conda deactivate

# end of script
```

For more detailed information about running your PBS job on the GPU nodes, see Requesting GPU Resources.

Requesting New or Missing Packages

If you need to use a specific package that is not currently installed in any of the environments, please send an email to support@nas.nasa.gov. In your email, provide the name of the package and request a ticket be opened with the Data Science team. We will help install the package into the existing environments or create a new environment if needed. You can also install your own packages or create your own conda environment.

You have two options to install your own Python packages in our machine learning environment:

- Use the pip tool to install them directly
- Build your own conda environment

Consider the benefits and disadvantages of each method, described below, before choosing which works best for you.

Using the pip Tool to Install Packages

Using the pip tool to install packages is easier than building your own conda environment and takes less space in your home directory.

The packages installed with pip will be applied to all environments that you load and will take precedence over the installed versions of packages within those environments—possibly causing version errors, depending on the differences. If the packages do not show up in other environments, make sure the Python versions in the loaded environment and the base environment match. Packages are Python-version-dependent; therefore, to ensure you can use the packages with the environment you want, load the environment before using pip.

The packages will be installed in the $HOME/.local/lib/... directory.

Complete the following steps to install packages on a Pleiades front-end system (PFE):

1. Load the miniconda3 module:

   ```
   pfe20 % module use -a /swbuild/analytix/tools/modulefiles
   pfe20 % module load miniconda3/v4
   ```

2. Use the pip command to install the packages:

   ```
   pfe20 % pip install --user package_name
   ```

Building Your Own Conda Environment

Building your own conda environment gives you the control to manage and install your own packages, and they will be less likely to have version errors than the pip-installed packages. The easiest way to create your own environment is to clone an existing conda environment into your own directory, then modify it.

Creating an environment can take up a significant portion of your disk quota, depending on the packages installed. To ensure that you can use your conda environment properly, please familiarize yourself with all the basic conda commands.

Before You Begin: Install a conda Token

The conda environments are supported under an Anaconda site license. In order to create a conda environment and be in compliance with the NASA Ames site license, we require all users to install a conda token prior to creating an environment. Complete these steps to acquire and install a token:

1. Request a NASA Ames conda token by sending email to: dataanalytics@nas.nasa.gov. You will receive an email, "Invitation to the NASA Ames organization on Anaconda Nucleus."
2. Create an Anaconda Nucleus account using the email you received. Click the **Join Organization** button and complete creating an account using the email address that the invite was sent to.
3. In two or three days, you will receive an email from Anaconda Nucleus with your new token and instructions on how to install it.

To activate your token on a NAS system:

1. Log on to a PFE.
2. Load the HECC Data Science miniconda module as follows:

3. ```
   module -a use /swbuild/analytix/tools/modulefiles
   module load miniconda3/v4
   ```

4. Run the conda token set command with token you received in the email:
   conda token set *token_string_from_email*
5. Verify the token is linked by using the conda info command. In the channel URLs, there should be:
   https://repo.anaconda.cloud/repo/main/...
   instead of:
   https://repo.anaconda.com/pkgs/main/...
   In addition, you can check whether https://repo.anaconda.cloud/repo/main appears in your .condarc file.

You should now have access to the commercial anaconda.cloud repo. Environments cloned from one of the Data Science conda environments or created from one of the Data Science YAML files will download and install packages from this repo.
   ```

If you have questions about this procedure or require assistance, please contact us by sending email to support@nas.nasa.gov.

Creating a Conda Environment from a Clone

Cloning from a YAML File

To build your own conda environment based on one of the NAS-provided conda environments, you can use the YAML file for the environment you want to clone. A list of YAML files for each of the NAS-provided conda environments can be found in the /swbuild/analytix/tools/miniconda3_220407_yml/ directory. For example:

- dask_mpi.yml
- horovod_env.yml
- jupyterlab_env.yml
- pyt1_10_2_env.yml
- pyt1_11_env.yml
- pyt1_12_env.yml
- pyt1_13_env.yml
- pyt1_8_env.yml
- r3_6_env.yml
- smartg_env.yml
- tf1_15_env.yml
- tf2_10_env.yml
- tf2_8_env.yml
- tf2_9_env.yml

For more information about YAML files, see the conda documentation.

Complete these steps to build a conda environment from a YAML file:

1. Load the miniconda3 module:

   ```
   pfe20 % module use -a /swbuild/analytix/tools/modulefiles
   pfe20 % module load miniconda3/v4
   ```

   TIP: If you don't want to fill up your home directory, you can put the environment in your /nobackup directory. To do this, set the CONDA_ENVS_PATH environment variable to point to a .conda/envs folder in /nobackup, as shown below. The folder is created the first time you run the conda command.
   - For bash: % export CONDA_ENVS_PATH=/nobackup/$USER/.conda/envs
   - For csh: % setenv CONDA_ENVS_PATH /nobackup/$USER/.conda/envs
2. Export the path to the target directory where you want to put the packages for installation:

   ```
   pfe20 % export CONDA_PKGS_DIRS=/nobackup/$USER/.conda/pkgs
   ```

3. Copy the YAML file you want to use to your local directory:

   pfe20 % cp /swbuild/analytix/tools/miniconda3_220407_yml/*filename*.yml *filename*.yml.

4. Open the file, then change the entry for name to a name for your new environment.

   Note: This step is required because keeping the default name may cause conflicts in cloning the environment.

5. At the prompt, enter the conda create command to create the environment:

   pfe20 % conda env create -f *filename*.yml

Cloning with the conda clone command

If the environment you want to clone is small , or you want to clone an environment based on one that is not provided by NAS, you can use the conda clone command to copy an existing environment to your local directory.

Complete these steps to build your own conda environment by cloning an existing environment:

1. Load the miniconda3 module:

   ```
   pfe20 % module use -a /swbuild/analytix/tools/modulefiles
   pfe20 % module load miniconda3/v4
   ```

2. Export the path to the target directory where you want to put the packages for cloning:

   pfe20 % export CONDA_PKGS_DIRS=/nobackup/$USER/*target_directory*

3. Clone an existing conda environment into a new folder. In this example, we create a new environment with no extra packages installed by cloning the base environment:

   pfe20 % conda create --name *my_env* --clone base

Your environment has been created.

Creating a Fresh Conda Environment

Another option is to create a fresh conda environment. Complete these steps to do so:

1. Load the miniconda3 module:

   ```
   pfe20 % module use -a /swbuild/analytix/tools/modulefiles
   pfe20 % module load miniconda3/v4
   ```

2. Export the path to the target directory where you want to put the packages for installation:

   ```
   pfe20 % export CONDA_PKGS_DIRS=/nobackup/$USER/.conda/pkgs
   ```

3. At the prompt, enter the conda create command to create the environment:

   pfe20 % conda create -n *my_env* python=3.x

Using Your Custom Environment

To use your custom environment, follow these steps:

1. Load the miniconda3 module:

   ```
   pfe20% module use -a /swbuild/analytix/tools/modulefiles
   pfe20% module load miniconda3/v4
   ```

2. Activate your environment:

   (for bash)
   pfe20% source activate *my_env*

   (for csh)
   pfe20% source /swbuild/analytix/tools/miniconda3_220407/bin/activate.csh *my_env*

   Note: If you created your environment in your /nobackup directory, use:

   (for bash)
   pfe20% export CONDA_ENVS_PATH=/nobackup/$USER/.conda/envs
   pfe20% source activate *my_env*

   (for csh)
   pfe20% setenv CONDA_ENVS_PATH /nobackup/$USER/.conda/envs
   pfe20% source /swbuild/analytix/tools/miniconda3_220407/bin/activate.csh *my_env*

3. Install any other packages you want by using the conda install command:

   *(my_env)*pfe20% conda install *package_name package_name* ...

Your newly installed packages are ready to use. After you finish using your environment, you can deactivate it by using theconda deactivate command.

If you have any issues, please contact us at support@nas.nasa.gov.

For more information about NAS-provided conda environments, see Using Conda Environments for Machine Learning.

The typical Jupyter installation includes more than 125 packages. If any one of these packages has been compromised, any account running Jupyter could become compromised. Keep in mind that even if the initial installation does not include any compromised packages, you might need to install additional packages—which could have been compromised. Or, you might run a Jupyter "notebook," obtained from an internet repository, that contains hidden malware; a Jupyter notebook has exactly the same access to files and other system resources as the user who started the notebook server. While we do not know of any active exploits against Jupyter at this time, this is a serious risk.

WARNING: We recommend you do not use Jupyter, or similar software, if your account has access to any ITAR, proprietary, or non-public data or programs.

Steps for Securely Setting Up Jupyter Notebook

Before you use Jupyter Notebook on NAS systems, you must complete these steps.

Notes:

- You only need to do these steps once (the first time you use Jupyter).
- Some of the command lines are too long to be formatted as one line, so they are broken with a backslash (\).

Before You Begin: Make sure you have set up SSH passthrough.

1. Log into a Pleiades front end (PFE) and load the miniconda3 module from the /swbuild/analytix directory:

   ```
   pfe20% module use -a /swbuild/analytix/tools/modulefiles
   pfe20% module load miniconda3/v4
   ```

2. To access Jupyter commands, activate a conda environment with Jupyter installed. All provided environments other than the base environment will work. For a list of other environments you can use, see Machine Learning: Overview. We suggest using the latest tensorflow or pytorch environment. In this example, we use the jupyterlab environment:

   ```
   (For bash) pfe20% source activate jupyterlab
   (For csh) pfe20% source /swbuild/analytix/tools/miniconda3_220407/bin/activate.csh jupyterlab
   ```

3. Set a password for your Jupyter notebooks, using both the jupyter notebook and the jupyter server commands:

   ```
   (jupyterlab)% jupyter notebook password
   (jupyterlab)% jupyter server password
   ```

   Two JSON password files will be written to your $HOME/.jupyter directory. Both commands are needed to ensure that the secure setup is compatible with older environments. In the future, Jupyter will move to using only jupyter server commands, so using both commands will also help guarantee future compatibility. The password used for both commands should be the same to reduce troubleshooting steps, but make sure the password follows the same security standards as other NAS passwords..

   Note: The default "token" authentication is not as secure as a password if you plan to share the notebook with others.

4. Make the Jupyter configuration information visible only to you:

   ```
   (jupyterlab)% chmod 700 $HOME/.jupyter
   ```

5. Protect Jupyter from request forgery by creating a self-signed certificate for your Jupyter HTTPS server:

   ```
   (jupyterlab)% openssl req -x509 -config \
   /swbuild/analytix/tools/jupyter_setup/jupyter_ssl.conf \
   -newkey rsa -out $HOME/.jupyter/jupyter.pem -days 1000
   ```

6. Copy the default NAS configuration files to your private configuration directory. These configuration files allow you to connect to the Jupyter server from your local web browser. The following command line copies two different configuration files (jupyter_server_config.py and jupyter_notebook_config.py) to your .jupyter directory:

   ```
   (jupyterlab)% cp /swbuild/analytix/tools/jupyter_setup/jupyter_*_config.py \
   $HOME/.jupyter/
   ```

7. Verify that the following files are present in your $HOME/.jupyter directory:

   ```
   (jupyterlab)% ls -l $HOME/.jupyter/
   -rw------- 1    103 Jun 17 14:35 jupyter_notebook_config.json
   -rw------- 1    103 Jun 17 14:35 jupyter_server_config.json
   -rw------- 1   1704 Jun 17 14:35 jupyter.key
   -rw------- 1   1155 Jun 17 14:35 jupyter.pem
   -rw------- 1  29875 Jun 17 14:35 jupyter_notebook_config.py
   -rw------- 1  57082 Jun 17 14:35 jupyter_server_config.py
   ```

   There might be additional files in the directory, but it must contain at least the six listed here.

8. Deactivate the environment:

   ```
   (jupyterlab)% conda deactivate
   ```

The setup is done. To clear any environment variables, log out of the PFE and log back in. You can now use Jupyter Notebook by following the instructions in Using Jupyter Notebook for Machine Learning Development.

Using Jupyter Notebook for Machine Learning Development on NAS Systems

Jupyter Notebook, an open-source application based on Interactive Python (IPython), is a useful tool for interactively exploring science data.

IMPORTANT: You must run Jupyter on a Pleiades front end (PFE) dedicated to you alone. You can do this either by running an interactive PBS session (for CPU or GPU jobs) or by reserving a dedicated compute node (for CPU jobs only). Running Jupyter on any other system, including an secure front end (SFE), Lou front end (LFE), or a non-dedicated PFE, is prohibited.

Steps for Using Jupyter Notebook for Machine Learning Development

Before You Begin: There are security issues to consider when running these tools on a multi-user system. You must complete the steps in Secure Set Up for Using Jupyter on NAS systems before you use Jupyter Notebook for the first time.

These steps describe how to use a PBS interactive job to set up a notebook running on one NVIDIA V100 GPU node. If you do not want to use a GPU node, you can reserve a dedicated compute node and use SSH to connect to the node. You can run Jupyter from any node type.

Notes:

- The steps use the jupyterlab conda environment as an example. For a list of other environments you can use, see Machine Learning: Overview.
- Some of the command lines are too long to be formatted as one line, so they are broken with a backslash (\).

To use Jupyter Notebook:

1. Log into a PFE and submit a PBS job:

   ```
   pfe20 % qsub -I -q v100@pbspl4 -lselect=1:ncpus=48:ngpus=1:model=cas_gpu:mem=300g \
   -l place=scatter:exclhost -l walltime=02:00:00
   ```

   Alternatively, if you do not want to use a GPU node, reserve a dedicated node and ssh into your reserved node:

   ```
   pfe20 % pbs_rfe --duration 10+ --model bro
   pfe20 % ssh nodename
   ```

2. On the compute node, load the miniconda3 module from the /swbuild/analytix directory:

   ```
   r101i2n6 % module use -a /swbuild/analytix/tools/modulefiles
   r101i2n6 % module load miniconda3/v4
   ```

   where r101i2n6 is the name of a node (the one you use will have a different name). Be sure to note the name of the compute node, as you will need it to access the Jupyter notebook on your local machine in step 5.

3. Activate a conda environment to use the proper libraries. For example, to activate the jupyterlab environment:
   - For bash: r101i2n6 % source activate jupyterlab
   - For csh: r101i2n6 % source \ /swbuild/analytix/tools/miniconda3_220407/bin/activate.csh jupyterlab

4. Start a Jupyter server that can be accessed by your local workstation:

   ```
   (jupyterlab)r101i2n6 % jupyter lab --no-browser
   ```

   If you get errors due to missing SSL files, try using the following (on one command line):

   ```
   (jupyterlab)r101i2n6 % jupyter lab --no-browser --certfile=~/.jupyter/jupyter.pem \
    --keyfile ~/.jupyter/jupyter.key
   ```

5. On your local workstation, open a terminal and type the following (on one command line) to connect by SSH to the compute node:

   ```
   your_local_system% ssh -l username -o "StrictHostKeyChecking ask" -L 8080:localhost:8888 \
   -o ProxyJump=sfe,pfe20 r101i2n6
   ```

   Note: You must specify the PFE you used to launch the compute node, and the name of the compute node. In this example, we specify pfe20 and r101i2n6. You might also need to specify the SFE. For *username*, use your NAS username. If you get a hostname error when trying to connect, try the command without including -l *username*.

6. You can now access Jupyter Lab from a browser on your workstation by connecting to https://localhost:8080/ or https://127.0.0.1:8080 You should see the Jupyter Lab landing page with a prompt for a password in your browser.

   Note: Currently, users are experiencing issues using some web browsers that do not allow you to use your self-signed certificate with Jupyter Lab on NAS systems. For this reason, we recommend that you use Firefox to access Jupyter Lab. If you encounter this issue while using Firefox, complete these steps to resolve the issue:

   1. Go to: **Firefox Settings > Privacy & Security**, then search for "certificates".
   2. Click **View Certificates** then click the **Servers** tab. Click **Add Exception** to add localhost server 127.0.0.1:8080. If the exception already exists, remove it then add it again.

   This should allow the notebook to access port 8080.

7. On the Jupyter Lab page, log in using the password you set in the Secure Setup procedure. You should now be able to access your files and launch an interactive notebook instance.

For more information about using this software, see Project Jupyter.

How to Set Up R Kernel in Jupyter Lab

NAS currently supports three versions of the R machine learning platform:

- R Versions 3.6 and 4.2 are provided in the r3_6 and r4_2 NAS conda environments, respectively.
- R Version 4.1 is provided as a NAS software package.

The versions in conda come preinstalled with many of the popular R packages used in data science and machine learning, including Jupyter Lab. Although the version provided as a NAS software package does not include Jupyter Lab, you can configure Jupyter Lab to work with this version of R by following the instructions below.

Before You Begin: Complete all the steps in the following articles:

- Secure Setup for Using Jupyter Notebook for Machine Learning Development on NAS Systems
- Using Jupyter Notebook for Machine Learning Development on NAS Systems

Complete the following steps to install R and set up Jupyter Lab to use it:

1. Load the following NetBSD Packages Collection (pkgsrc) module and start R:

   ```
   pfe27% module load pkgsrc/2022Q1-rome
   pfe27% R
   ```

2. Install IRkernel in R:

   ```
   > install.packages("IRkernel")
   ```

3. Run IRkernel in R:

   ```
   > IRkernel::installspec()
   ```

4. Copy the IRkernel directory to your Jupyter data directory. For example, if your IRkernel directory is in$HOME/R/x86_64-redhat-linux-gnu-library/4.1, do:

   ```
   > cp -R $HOME/R/x86_64-redhat-linux-gnu-library/4.1/IRkernel $HOME/.local/share/jupyter
   ```

5. Change the R path (the first "R") in the $HOME/.local/share/jupyter/IRkernel/kernelspec /kernel.json file to: /nasa/pkgsrc/toss4/2022Q1-rome/bin/R *(or other R)*
6. Load Jupyter and run R.

Using TensorBoard for Machine Learning Development on NAS Systems

TensorBoard is a tool for providing the measurements and visualizations needed during the machine learning workflow. It enables tracking of experiment metrics such as loss and accuracy, visualizing the model graph, projecting embeddings to a lower dimensional space, and much more.

This article explains how to quickly get started with TensorBoard. You will need to load and activate a NAS conda environment that includes TensorFlow.

Before You Begin: In order to use the methods described in this article, you must set up SSH Passthrough.

Creating Data for TensorBoard

Within your TensorFlow application there is a piece of code that does the model fitting, as shown below. Here, we create a TensorBoard callback:

```
log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback= tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)
```

Add the argument callbacks=[tensorboard_callback]), which accesses the function:

```
model.fit(train_ds,
    epochs=EPOCHS,
    steps_per_epoch=int(image_count/BATCH),
    verbose=verbose,
    callbacks=[tensorboard_callback])
```

Once your application is finished, you will see a directory called logs/.

Using TensorBoard

Complete these steps to use TensorBoard.

1. Log into a Pleiades front end (PFE) and reserve a compute node. In this example, we use pfe20:

   ```
   pfe20 % pbs_rfe --duration 10+ --model bro
   ```

   Alternatively, you can submit a PBS job:

   ```
   pfe20 % qsub -I -q devel -lselect=1:ncpus=16:model=bro,walltime=1:00:00
   ```

   Note: You don't need a GPU node to look at the TensorBoard data.

2. On the compute node, load the miniconda3 module from the /swbuild/analytix directory:

   ```
   r601i1n1 % module use -a /swbuild/analytix/tools/modulefiles
   r601i1n1 % module load miniconda3/v4
   ```

   where r601i1n1 is the name of a compute node (the one you use will have a different name). Be sure to note the name of the compute node, as you will need it to access the TensorBoard on your local machine in step 5.

3. Activate a conda environment to use the proper libraries. For example, where *env_name* is the name of the environment:
   - For bash:
     r601i1n1 % source activate *env_name*
   - For csh (on one line):
     r601i1n1 % source /swbuild/analytix/tools/miniconda3_220407/bin/activate.csh *env_name*

4. Start a TensorBoard on the compute node:

   ```
   (env_name)r601i1n1 % tensorboard --logdir=logs
   Serving TensorBoard on localhost; to expose to the network,
   use a proxy or pass --bind_all
   TensorBoard 2.2.1 at http://localhost:6006/ (Press CTRL+C to quit)
   ```

5. On your local workstation, open a terminal and type the following (on one command line) to connect via SSH to the compute node:

   ```
   your_local_system% ssh -o "StrictHostKeyChecking ask" -L
   8080:localhost:6006 -o ProxyJump=sfe,pfe20 r601i1n1
   ```

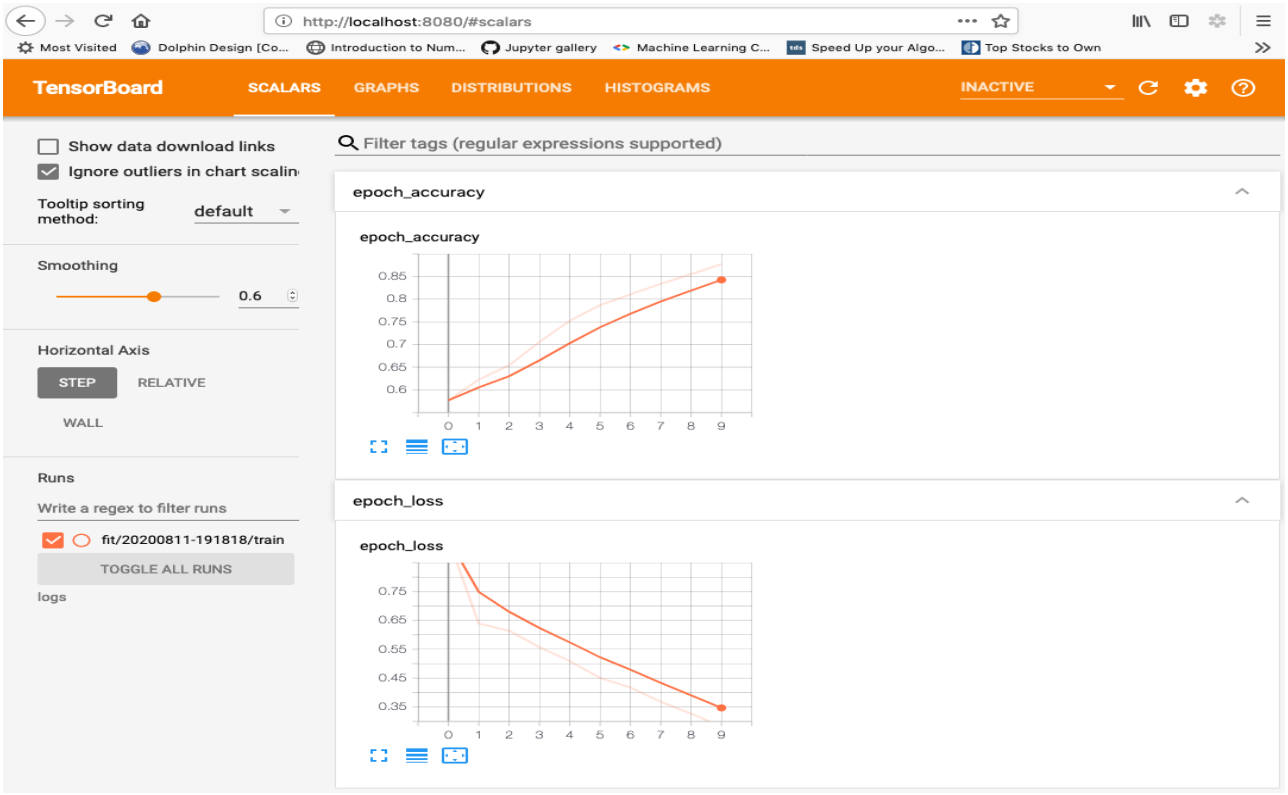
   Note: You must specify the PFE you used to launch the compute node, and the name of the compute node. In this example, we specify pfe20 and r601i1n1.

6. You can now access Jupyter Lab from a browser on your workstation by connecting to:

   http://localhost:8080/ or http://127.0.0.1:8080

   Note: If your browser objects to the self-signed certificate used by this instance of TensorBoard, allow it to continue.

Once you have completed these steps, you will get this interactive screen to create customized measurements and views of your data:



References

For more information, see the following documentation:

- Get Started with TensorBoard
- TensorBoard Tutorial

TensorFlow uses strategies to make distributing neural networks across multiple devices easier. The strategy used to distribute TensorFlow across multiple nodes is multiworkermirroredstrategy, which is slightly more complicated to implement than other strategies like mirroredstrategy.

Using MultiWorkerMirroredStrategy

The basic steps to use multiworkeredmirroredstrategy on HECC resources are:

1. Request multiple nodes in your PBS script.
2. Create an .sh script that can load the miniconda module and run the Python script with proper command line inputs.
3. Set the TF_CONFIG environment variable in the .py file to point to the other nodes that you have requested and chose a port to use for communication.
4. Use tf.data.shard() to make sure that that training data is properly distributed between each node.
5. Use ssh to log into each node individually and run the .sh script, passing the node information to each node to run as a background process.
6. Run the chief worker node after all the other nodes have been started.

For more information on this strategy, see TensorFlow's website: Distributed Training with TensorFlow and Multi-Worker Training with Keras.

Notes:

- This strategy is still considered 'experimental' by TensorFlow, so the usage might not be stable.
- Currently, the method is scalable for CPU nodes, but not GPU nodes.
- Sleeping in the for loop is used to help guarantee that the worker nodes are started and waiting for the chief node. If the chief node starts before the worker nodes, the code might hang.

We have prepared the following three files to demonstrate the method of running multiple CPU nodes over TensorFlow:

PBS Script
: Requests resources and calls an .sh file.

Shell Script (run_vgg.sh)
: Starts the TensorFlow Python code.

Python Script (vgg_dist.py)
: Sample Python code to perform training in TensorFlow.

Each of these files is shown below.

PBS Script

Note: This script has been tested using bash and may require changes in order to work with csh. Please contact Support if you encounter any issues.

```
#!/bin/bash
#PBS -l select=3:ncpus=16:model=bro
#PBS -q normal
#PBS -l walltime=1:00:00
#PBS -j oe
#PBS -o node_3_test.log
#PBS -N multinodetest

#change to nobackup
#cd $PBS_O_WORKDIR
BASE=$(pwd)

echo "The base node is:"
hostname
printf "\n\n"

#get node information from nodefile
NODES=($( cat $PBS_NODEFILE | uniq ))
NUM_OF_NODES=${#NODES[@]}
#worker number for other nodes
C=1

# for each node that's not the current node
for node in ${NODES[@]}
do
  if [[ $node != $(eval hostname) ]]
  then
    # ssh into each node and run the .sh script with node info
    # run in background
    ssh $node "$BASE/run_vgg.sh $C ${NODES[*]}" &
```

```
    C=$(($C+1))
    sleep 2
  fi
done

#run the main worker node
$BASE/run_vgg.sh 0 ${NODES[*]}

echo "Done with PBS"
```

run_vgg.sh Script

```
#!/bin/bash
#load module command
source /usr/local/lib/global.profile

#cd to code location
cd /nobackup/$USER
BASE=/nobackup/$USER/vgg_dist

echo "On node: "
hostname

module purge
module -a use /swbuild/analytix/tools/modulefiles
module load miniconda3/v4
source activate conda_environment

echo "loaded conda env"

# Set environments for CPU
setenv KMP_AFFINITY "granularity=fine,verbose,compact,1,0"
setenv KMP_SETTINGS 1
setenv OMP_NUM_THREADS 16     # Sandy has 16 CPUs per node
setenv KMP_BLOCKTIME 30

#run python script with inputs from this .sh script
python $BASE/vgg_dist.py $@
conda deactivate
```

vgg_dist.py Script

Note: Three of the lines in this script are too long to be formatted as one line, so they are broken with a back slash (\).

```
import os
import json
import sys
import time
import pathlib
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import Conv2D, AveragePooling2D, \
MaxPool2D, Flatten, Dense, BatchNormalization, Dropout, Activation
#set seeds for testing
tf.random.set_seed(22)
base_dir = os.path.dirname(os.path.abspath(__file__))
# ---------- Set up TF_CONFIG ---------- #
# load the index and node info from the command line
index = int(sys.argv[1])
verbose = index < 1
nodes=[]
# node names for each node, append a port # to the names
for i in range(2,len(sys.argv)):
    nodes.append(sys.argv[i] + ':2001')
# set TF_CONFIG variable that MultiWorkerMirroredStrategy needs
os.environ['TF_CONFIG'] = json.dumps({
    'cluster': {
        'worker': nodes
    },
    'task': {'type': 'worker', 'index': index}
})
print(os.environ['TF_CONFIG'])
'''
verbose = 1
index = 0
'''
#if(verbose):
    #tf.debugging.set_log_device_placement(True)
```

```
#set up strategies
#strategy = tf.distribute.OneDeviceStrategy(device='/device:CPU:0')
strategy = tf.distribute.experimental.MultiWorkerMirroredStrategy \
(tf.distribute.experimental.CollectiveCommunication.AUTO)
#strategy = tf.distribute.MirroredStrategy()
EPOCHS = 5
#batch size per worker on each shard of the dataset
BATCH = 256
LEARNING_RATE = 0.001
IMG_WIDTH, IMG_HEIGHT = 224, 224
AUTOTUNE = tf.data.experimental.AUTOTUNE
train_dir = base_dir + '/data/train'
train_dir = pathlib.Path(train_dir)
image_count = len(list(train_dir.glob('*/*.jpg')))
CLASS_NAMES = np.array([item.name for item in train_dir.glob('*') \
if item.name != "LICENSE.txt"])
def get_label(file_path):
    # convert the path to a list of path components
    parts = tf.strings.split(file_path, '/')
    # The second to last is the class-directory
    return parts[-2] == CLASS_NAMES
def decode_img(img):
    # convert the compressed string to a 3D uint8 tensor
    img = tf.image.decode_jpeg(img, channels=3)
    # Use `convert_image_dtype` to convert to floats in the [0,1] range.
    img = tf.image.convert_image_dtype(img, tf.float32)
    # resize the image to the desired size.
    return tf.image.resize(img, [IMG_WIDTH, IMG_HEIGHT])
def process_path(file_path):
    label = get_label(file_path)
    # load the raw data from the file as a string
    img = tf.io.read_file(file_path)
    img = decode_img(img)
    return img, label
def prepare_for_training(ds, cache=True, shuffle_buffer_size=1000, \
num_workers=1, index=0):
    # This is a small dataset, only load it once, and
    # keep it in memory.
    # use `.cache(filename)` to cache preprocessing work for
    # datasets that don't fit in memory.
    ds.shard(num_workers, index)
    if cache:
        if isinstance(cache, str):
            ds = ds.cache(cache)
        else:
            ds = ds.cache()
    ds = ds.shuffle(buffer_size=shuffle_buffer_size)
    # Repeat forever
    ds = ds.repeat()
    ds = ds.batch(BATCH)
    # `prefetch` lets the dataset fetch batches in the background while
    # the model is training.
    ds = ds.prefetch(buffer_size=AUTOTUNE)
    return ds
#list_ds = tf.data.Dataset.list_files(str(train_dir/'*/*'))
#labeled_ds = list_ds.map(process_path, num_parallel_calls=AUTOTUNE)
#train_ds = prepare_for_training(labeled_ds, num_workers=len(nodes), \
# index=index)
# imagegen = ImageDataGenerator(rescale=1./255)
# train = imagegen.flow_from_directory(directory='./data/train', \
# batch_size=BATCH, target_size=(224,224))
# val = imagegen.flow_from_directory(directory='./data/val', batch_size=BATCH, \
#target_size=(224,224))
def build_vgg_a():
    model = keras.Sequential()
    model.add(Conv2D(input_shape=(224,224,3),filters=64,kernel_size=(3,3), \
padding="same", activation="relu"))
    model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))
    model.add(Conv2D(filters=128, kernel_size=(3,3), padding="same", activation="relu"))
    model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))
    model.add(Conv2D(filters=256, kernel_size=(3,3), padding="same", activation="relu"))
    model.add(Conv2D(filters=256, kernel_size=(3,3), padding="same", activation="relu"))
    model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))
    model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
    model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
    model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))
    model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
    model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
    model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))
    model.add(Flatten())
    # added batch norm to keep weights down
```

```
    model.add(Dense(units=4096))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(Dense(units=4096))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(Dense(units=2, activation="softmax"))
    return model
#use the distributed strategy
with strategy.scope():
    list_ds = tf.data.Dataset.list_files(str(train_dir/'*/*'))
    labeled_ds = list_ds.map(process_path, num_parallel_calls=AUTOTUNE)
    print(strategy.num_replicas_in_sync)
    train_ds = prepare_for_training(labeled_ds, \
num_workers=strategy.num_replicas_in_sync, index=index)
    opt = keras.optimizers.Adam(lr = LEARNING_RATE)
    model = build_vgg_a()
    #compile model
    model.compile(optimizer= opt,
        loss=keras.losses.categorical_crossentropy,
        metrics=['accuracy']
        )
if(verbose):
    model.summary()
start = time.time()
model.fit(train_ds,
    epochs=EPOCHS,
    steps_per_epoch=int(image_count/BATCH),
    verbose=verbose
    )
if(verbose):
  print('took ' + str((time.time()-start)/60) + ' minutes'
```

Sample PBS Scripts for Using PyTorch and TensorFlow

PyTorch and TensorFlow are available in NAS-provided environments, which are listed in the Machine Learning Overview. You can activate the environments in interactive mode or in a PBS script.

The examples provided in this article demonstrate how to load the miniconda module, activate an environment, run your program, and deactivate the environment in a PBS script.

To help you determine which environment you want to activate, you can view all of the available environments by using the following conda command:

```
% conda info --envs
```

Sample PBS Scripts for PyTorch

Four sample PBS scripts are provided for PyTorch: bash and csh scripts for CPUs, and bash and csh scripts for GPUs.

Using PyTorch with CPUs

These sample PBS scripts request Broadwell CPUs and engage the PyTorch environment pyt1_12.

bash script for using PyTorch with CPUs:

```
##!/bin/bash
##This script test different conda environments running on Broadwell Nodes
#PBS -l select=1:ncpus=16:model=bro
#PBS -l site=static_broadwell
#PBS -q devel
#PBS -l walltime=02:00:00
#PBS -j oe
#PBS -N model_bro-check
#PBS -W group_list=n1853


#load analytix modules
module use -a /swbuild/analytix/tools/modulefiles
module load miniconda3/v4


#activate pyt1_12 pytorch conda environment
source activate pyt1_12


##This script trains a neural network model on MNIST characters
cd /home7/analytix/examples/pytorch/src/main/python
echo "  "
echo "  "
echo "pytorch mnist test"
echo "  "
echo "  "
python3 torch_mnist_benchj.py
echo "  "
echo "  "

#exit environment
conda deactivate
```

csh script for using PyTorch with CPUs:

```
#!/bin/csh -x
##This script test different conda environments running on Broadwell Nodes
#PBS -l select=1:ncpus=16:model=bro
#PBS -l site=static_broadwell
#PBS -q devel
#PBS -l walltime=02:00:00
#PBS -j oe
#PBS -N model_bro-check
#PBS -W group_list=n1853


#load analytix modules
module use -a /swbuild/analytix/tools/modulefiles
module load miniconda3/v4
```

```
#activate pyt1_12 pytorch conda environment
source /swbuild/analytix/tools/miniconda3_220407/bin/activate.csh pyt1_12


##This script trains a neural network model on MNIST characters
cd /home7/analytix/examples/pytorch/src/main/python
echo "  "
echo "  "
echo "pytorch mnist test"
echo "  "
echo "  "
python3 torch_mnist_benchj.py
echo "  "
echo "  "

#exit environment
conda deactivate
```

Using PyTorch with GPUs

These sample PBS scripts request Cascade Lake GPU nodes and engage the PyTorch environment pyt1_12.

bash script for using PyTorch with GPUs:

```
##!/bin/bash
##This script test different conda environments running on Cascade Lake GPU Nodes
#PBS -q v100@pbspl4
#PBS -W block=true
#PBS -lselect=1:ncpus=48:ngpus=4:model=cas_gpu:mem=300g
#PBS -l place=scatter:exclhost
#PBS -l walltime=02:00:00
#PBS -j oe
#PBS -N model_cas_gpu-check
#PBS -W group_list=n1853


#load analytix modules
module use -a /swbuild/analytix/tools/modulefiles
module load miniconda3/v4


#activate pyt1_12 pytorch conda environment
source activate pyt1_12


##This script trains a neural network model on MNIST characters
cd /home7/analytix/examples/pytorch/src/main/python
echo "  "
echo "  "
echo "pytorch mnist test"
echo "  "
echo "  "
python3 torch_mnist_benchj.py
echo "  "
echo "  "

#exit environment
conda deactivate
```

csh script for using PyTorch with GPUs:

```
#!/bin/csh -x
##This script test different conda environments running on Cascade Lake GPU Nodes
#PBS -q v100@pbspl4
#PBS -W block=true
#PBS -lselect=1:ncpus=48:ngpus=4:model=cas_gpu:mem=300g
#PBS -l place=scatter:exclhost
#PBS -l walltime=02:00:00
#PBS -j oe
#PBS -N model_cas_gpu-check
#PBS -W group_list=n1853


#load analytix modules
module use -a /swbuild/analytix/tools/modulefiles
module load miniconda3/v4
```

```
#activate pyt1_12 pytorch conda environment
source /swbuild/analytix/tools/miniconda3_220407/bin/activate.csh pyt1_12


##This script trains a neural network model on MNIST characters
cd /home7/analytix/examples/pytorch/src/main/python
echo "  "
echo "  "
echo "pytorch mnist test"
echo "  "
echo "  "
python3 torch_mnist_benchj.py
echo "  "
echo "  "

#exit environment
conda deactivate
```

Sample PBS Scripts for TensorFlow

Four sample PBS scripts are provided for TensorFlow: bash and csh scripts for CPUs, and bash and csh scripts for GPUs.

Using TensorFlow with CPUs

These sample scripts request Broadwell CPUs and engage the TensorFlow environment tf2_9.

bash script for using TensorFlow with CPUs:

```
##!/bin/bash
##This script test different conda environments running on Broadwell Nodes
#PBS -l select=1:ncpus=16:model=bro
#PBS -l site=static_broadwell
#PBS -q devel
#PBS -l walltime=02:00:00
#PBS -j oe
#PBS -N model_bro-check
#PBS -W group_list=n1853


#load analytix modules
module use -a /swbuild/analytix/tools/modulefiles
module load miniconda3/v4


#activate tf2_9 tensorflow conda environment
source activate tf2_9


##This script trains a neural network model on MNIST characters
cd /home7/analytix/examples/mnist/src/main/python
echo "  "
echo "  "
echo "tensorflow mnist test"
echo "  "
echo "  "
python3 mnist.py
echo "  "
echo "  "

#exit environment
conda deactivate
```

csh script for using TensorFlow with CPUs:

```
#!/bin/csh -x
##This script test different conda environments running on Broadwell Nodes
#PBS -l select=1:ncpus=16:model=bro
#PBS -l site=static_broadwell
#PBS -q devel
#PBS -l walltime=02:00:00
#PBS -j oe
#PBS -N model_bro-check
#PBS -W group_list=n1853


#load analytix modules
```

```
module use -a /swbuild/analytix/tools/modulefiles
module load miniconda3/v4


#activate tf2_9 tensorflow conda environment
source /swbuild/analytix/tools/miniconda3_220407/bin/activate.csh tf2_9


##This script trains a neural network model on MNIST characters
cd /home7/analytix/examples/mnist/src/main/python
echo "  "
echo "  "
echo "tensorflow mnist test"
echo "  "
echo "  "
python3 mnist.py
echo "  "
echo "  "


#exit environment
conda deactivate
```

Using TensorFlow with GPUs

These sample scripts request Cascade Lake GPU nodes and engage the TensorFlow environment `tf2_9`.

bash script for using TensorFlow with GPUs:

```
##!/bin/bash
##This script test different conda environments running on Cascade Lake GPU Nodes
#PBS -q v100@pbspl4
#PBS -W block=true
#PBS -lselect=1:ncpus=48:ngpus=4:model=cas_gpu:mem=300g
#PBS -l place=scatter:exclhost
#PBS -l walltime=02:00:00
#PBS -j oe
#PBS -N model_cas_gpu-check
#PBS -W group_list=n1853


#load analytix modules
module use -a /swbuild/analytix/tools/modulefiles
module load miniconda3/v4


#activate tf2_9 tensorflow conda environment
source activate tf2_9


##This script trains a neural network model on MNIST characters
cd /home7/analytix/examples/mnist/src/main/python
echo "  "
echo "  "
echo "tensorflow mnist test"
echo "  "
echo "  "
python3 mnist.py
echo "  "
echo "  "

#exit environment
conda deactivate
```

csh script for using TensorFlow with GPUs:

```
#!/bin/csh -x
##This script test different conda environments running on Cascade Lake GPU Nodes
#PBS -q v100@pbspl4
#PBS -W block=true
#PBS -lselect=1:ncpus=48:ngpus=4:model=cas_gpu:mem=300g
#PBS -l place=scatter:exclhost
#PBS -l walltime=02:00:00
#PBS -j oe
#PBS -N model_cas_gpu-check
#PBS -W group_list=n1853
```

```
#load analytix modules
module use -a /swbuild/analytix/tools/modulefiles
module load miniconda3/v4


#activate tf2_9 tensorflow conda environment
source /swbuild/analytix/tools/miniconda3_220407/bin/activate.csh tf2_9


##This script trains a neural network model on MNIST characters
cd /home7/analytix/examples/mnist/src/main/python
echo "  "
echo "  "
echo "tensorflow mnist test"
echo "  "
echo "  "
python3 mnist.py
echo "  "
echo "  "


#exit environment
conda deactivate
```

Using Horovod for Distributed Training

Horovod is a Python package hosted by the LF AI and Data Foundation, a project of the Linux Foundation. You can use it with TensorFlow and PyTorch to facilitate distributed deep learning training. Horovod is designed to be faster and easier to use than the built-in distribution strategies that TensorFlow and PyTorch use. It uses MPI as the communication layer, rather than the parameter servers used by the built-in strategies.

For more information about Horovod, see:

- Horovod website
- Horovod on GitHub

NAS provides a conda environment that enables you to use Horovod on NAS compute resources. The basic steps are:

- Request multiple GPU nodes.
- Load the miniconda and module load mpi-hpe/mpt modules.
- Activate the NAS-provided horovod environment.
- Run a script that sets the proper environment variables (a sample script is provided below).
- Use mpiexec to run Python code that has the proper Horovod integrations (also provided below).

You can find a full working example of Horovod in the /swbuild/analytix/examples/horovod directory on Pleiades. The example includes a PBS script; the set_vars.sh script, which sets environment variables; a working example of TensorFlow and Keras ResNet; and a working example of PyTorch ResNet.

PBS Request

The workflow for the PBS request is outlined here:

1. Set mpiprocs equal to the number of GPUs requested per node.
2. Set place=scatter:excl. (Note that although other options might work, this option is certain to work.)
3. Load the miniconda module from the /swbuild/analytix/ directory on Pleiades, and the module load mpi-hpe/mpt module.
4. Load the horovod environment.
5. Set the proper environment variables on all nodes by running the set_vars.sh script (or similar).
6. Use mpiexec to run the Python code and set the number of ranks equal to the number of nodes times the number of GPUs requested.
7. Use the -bind-to none and -map-by slot arguments to help improve performance.
8. Optionally, use the -prepend-rank argument to help debug and keep track of each process.
9. After the MPI process finishes, deactivate the environment to clean up.

Sample PBS Script

This script is also provided in the /swbuild/analytix/examples/horovod directory.

```
#PBS -l select=2:model=cas_gpu:mpiprocs=2:ncpus=8:ngpus=2:mem=32g
#PBS -l place=scatter:shared
#PBS -q v100@pbspl4

#PBS -l walltime=1:00:00
#PBS -N hvcas_2_node_2_gpu_test
#PBS -j oe
#PBS -o horovod_cas_test.log


BASE=$(pwd)

SEVERAL_TRIES=/u/scicon/tools/bin/several_tries

# load modules
module -a use /swbuild/analytix/tools/modulefiles
module load miniconda3/v4
module load mpi-hpe/mpt

# load env
source activate horovod

# set variables on this node and other nodes
# sets CUDA_VISIBLE_DEVICES and NCCL_IB_DISABLE on all requested nodes
source set_vars.sh 2

# run horovod benchmark test
# np equal to the number of gpus * number of nodes
# in this case, there are 2 nodes with 2 GPUs each

#several tries script to get it to work
```

```
source $SEVERAL_TRIES \
mpiexec -p "[%g]" -stats -v \
  -np 4 \
  python python_code.py


conda deactivate
echo "Done with PBS"
```

Environment Variables

The NAS-provided set_vars.sh script manages the environment variables needed for Horovod to work properly on all nodes. The script sets the following variables:

NCCL_IB_DISABLE
: This variable must be set to 1, on the main node only, to allow NCCL to communicate between nodes.

CUDA_VISIBLE_DEVICES
: This variable must be set on each node as a list of identifiers for each GPU used on the node. The script gives each GPU a number, but the identifier can be anything provided that the GPUs on the same node have different identifiers.

GPU_LIST
: Sets CUDA_VISIBLE_DEVICES to ints to let the other nodes use the GPUs when they are activated by MPI. The default is 0.

MPI_LAUNCH_TIMEOUT
: Set MPI_LAUNCH_TIMEOUT to longer values so horovod has time to load. The default value is 60.

MPI_UNBUFFERED_STDIO
: Set std out buffer to TRUE to have no limit.

NCCL_DEBUG
: Set to INFO to use debugging features; helps print which nodes are used in code.

The set_vars.sh Script

This script is also provided in the /swbuild/analytix/examples/horovod directory.

```
#!/bin/bash
# usage:
# set_vars.sh <num of GPUs per node>
# defaults to 1 GPU if not specified

# functionality
# renames the GPUS on nodes so NCCL recognizes them,
sets environment variables to have NCCL work properly and output debugging logs

echo "Im on $(hostname)"

# set MPI_LAUNCH_TIMEOUT to longer values so horovod has time to load
export MPI_LAUNCH_TIMEOUT=60
# set std out buffer to have no limit
export MPI_UNBUFFERED_STDIO=true

#set NCCL to use ip-over-infiniband rather than default RoCE,
#since our infiniband does not have RoCE export NCCL_IB_DISABLE=1

#used for debugging, helps print which nodes are used in code
#export NCCL_DEBUG=INFO

#set CUDA_VISIBLE_DEVICES to ints to let the other nodes use the GPUS
#when they are activated by mpi

GPU_LIST=0

for ((i=1; i<=($1-1); i++));
do
  GPU_LIST=$GPU_LIST,$i
done

echo "numbers of GPUs used: $GPU_LIST"
export CUDA_VISIBLE_DEVICES=$GPU_LIST

#set CUDA_VISIBLE_DEVICES on other nodes
NODES=($( cat $PBS_NODEFILE | sort | uniq ))
NUM_OF_NODES=${#NODES[@]}

for node in ${NODES[@]}
do
  if [[ $node != $(eval hostname) ]]
  then
    ssh $node echo "Im on $node"; export CUDA_VISIBLE_DEVICES=$GPU_LIST
  fi
done
```

Python Code Integrations

Your Python code also needs the proper Horovod integrations to work. The following basic additions are required:

- Run hvd.init().
- Pin each GPU to a single process.
- Scale learning rate by the number of workers.
- Use a hvd.DistibutedOptimizer().
- Synchronize the worker's initial weights.
- Save checkpoints and models only on the main worker.

Sample Python stubs are provided below. You can also find more information at the following locations on GitHub:

- Detailed information and code examples
- Specific information for TensorFlow

Sample Python Stub

The following sample Python stub contains the TensorFlow integrations that are required for use with Horovod.

Sample Python Stub with TensorFlow Integrations

```
import tensorflow as tf
import horovod.tensorflow.keras as hvd

# Horovod: initialize Horovod.
hvd.init()

# Horovod: pin GPU to be used to process local rank (one GPU per process)
gpus = tf.config.experimental.list_physical_devices('GPU')
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)
if gpus:
    tf.config.experimental.set_visible_devices(gpus[hvd.local_rank()], 'GPU')

# Scale learning rate by horovod size
lr = 0.01 * hvd.size()

# Set up your model and chosen optimizer
model = ...
opt = tf.optimizers.SGD(lr)

# Set up data
data = ...
target = ...
dataset = tf.data.Dataset.from_tensor_slices((data, target)).cache().repeat().batch(batch_size)


fp16_allreduce = False
# Horovod: (optional) compression algorithm.
compression = hvd.Compression.fp16 if fp16_allreduce else hvd.Compression.none
# Horovod: add Horovod DistributedOptimizer.
opt = hvd.DistributedOptimizer(opt, compression=compression)

# Horovod: Specify `experimental_run_tf_function=False` to ensure TensorFlow
# uses hvd.DistributedOptimizer() to compute gradients.
model.compile(loss=tf.losses.SparseCategoricalCrossentropy(),
              optimizer=opt,
              experimental_run_tf_function=False)

callbacks = [
    # Horovod: broadcast initial variable states from rank 0 to all other processes.
    # This is necessary to ensure consistent initialization of all workers when
    # training is started with random weights or restored from a checkpoint.
    hvd.callbacks.BroadcastGlobalVariablesCallback(0),

    # Horovod: average metrics among workers at the end of every epoch.
    #
    # Note: This callback must be in the list before the ReduceLROnPlateau,
    # TensorBoard or other metrics-based callbacks.
    hvd.callbacks.MetricAverageCallback(),
]


# Horovod: save checkpoints only on worker 0.
if hvd.rank() == 0:
    checkpoint_cb = tf.keras.callbacks.ModelCheckpoint(...)
    callbacks.append(checkpoint_cb)
```

```
# Train the model.
model.fit(
    dataset,
    batch_size=batch_size,
    steps_per_epoch=num_batches_per_iter,
    callbacks=callbacks,
    epochs=num_iters,
    verbose=0,
)
```

Using GPU Nodes

GPU Node Reservation: Thirty Cascade Lake-V100 GPU nodes (cas_gpu) are currently reserved for a special project. During this time, we recommend using the Skylake-V100 GPU nodes (sky_gpu) for your jobs requiring V100 GPUs.

A graphics processing unit (GPU) is a hardware device that can accelerate some computer codes and algorithms. This article describes the two types of GPU nodes currently available at NAS. For information about running your PBS jobs on the GPU nodes, see Requesting GPU Resources. For information about developing code for the GPU nodes, see GPU Programming.

For more general information on GPUs, see the GPGPU article on Wikipedia and the NVIDIA website, which has information for developers.

If you have an application that can take advantage of GPU technology, you can use one of the NAS GPU models:

- model=sky_gpu

  19 nodes total. There are two different configurations:

  The Skylake sockets used in the sky_gpu nodes are Intel Xeon Gold 6154 while those in the Electra Skylake nodes are Intel Xeon Gold 6148. The main differences in these two configurations (sky_gpu vs. Electra Skylake) are: (1) number of cores - 18 vs. 20; (2) CPU clock speed - 3.0 vs. 2.4 GHz; (3) L3 cache size - 24.75 vs. 27.5 MB; (4) memory size - 384 vs. 192 GB; and (5) number of UPI links - 3 vs. 2.

  * Three of the 17 nodes are reserved for a special project; 14 nodes are available for use.

    - 17* nodes, each containing two Skylake CPU sockets and **four** NVIDIA Volta V100 GPU cards, where the CPUs and GPUs are connected via PCI Express bus
    - Two nodes, each containing two Skylake CPU sockets and **eight** NVIDIA Volta V100 GPU cards, where the CPUs and GPUs are connected via PCI Express bus
- model=cas_gpu

  38* nodes total. Each node contains two Cascade Lake CPU sockets and four NVIDIA Volta V100 GPU cards, where the CPUs and GPUs are connected via PCI Express bus.

  The Cascade Lake sockets used in the cas_gpu nodes are Intel Xeon Platinum 8268 while those in the Aitken Cascade Lake nodes are Intel Xeon Gold 6248. The main differences in these two configurations (cas_gpu vs Aitken Cascade Lake) are: (1) number of cores - 24 vs 20; (2) CPU clock speed - 2.9 vs. 2.5 GHz; (3) L3 cache size - 35.75 vs. 27.5 MB; and (4) memory size (384 vs 192 GB).

  * Four of the 38 nodes are reserved for a special project. 34 nodes are available for public use.

The SBU rates for the GPU nodes are:

- sky_gpu with four V100s: 9.82
- sky_gpu with eight V100s: 15.55
- cas_gpu with four V100s: 9.82

	Node Details	
	sky_gpu	**cas_gpu**
Architecture	Apollo 6500 Gen10	Apollo 6500 Gen10
Total # of Nodes	19	38
Host name	r101i0n[0-11,14-15], r101i1n[0-2] and r101i0n[12-13]*	r101i2n[0-17], r101i3n[0-15] and r101i4n[0-3]
	CPU Host	
Processors Model	18-core Xeon Gold 6154	24-core Xeon Platinum 8268
# of CPU Cores/Node	36	48
CPU-Clock	3.0 GHz	2.9 GHz
Maximum Double Precision Floating Point Operations per cycle per core	32	32
Total CPU Double Precision Flops/Node	3,456 GFlops	4,454 GFlops
Memory/Node	384 GB (DDR4)	384 GB (DDR4)
	GPU	
Device Name	Tesla V100-SXM2-32GB	Tesla V100-SXM2-32GB
Clock Rate (Base/Boost)	1290 MHz/1530 MHz	1290 MHz/1530 MHz
# of coprocessors/Node	4 (17 nodes) or 8 (2 nodes)	4
# of Single Precision CUDA Cores/coprocessor	5120	5120
# of Double Precision CUDA Cores/coprocessor	2560	2560
# of Tensor Cores/coprocessor	640	640

Total GPU Double Precision Flops/coprocessor (Base/Boost)	6.6 TFlops/7.8 TFlops	6.6 TFlops/7.8 TFlops
Total GPU Double Precision Flops/Node (Base/Boost)	26.4 TFlops/31.2 TFlops for 4 V100 and 52.8 TFlops/62.4 TFlops for 8 V100	26.4 TFlops/31.2 TFlops
L2 Cache	6144 KB	6144 KB
Global Memory/coprocessor	32 GB (HBM2)	32 GB (HBM2)
Memory Clock Rate	877 MHz	877 MHz
Memory Bus Width	4096 bits	4096 bits
Memory Bandwidth	900 GB/s	900 GB/s
GPU-to-GPU Communication	SXM2 mezzanine connector with NVLink 2.0 with 300 GB/s bi-directional bandwidth	SXM2 mezzanine connector with NVLink 2.0 with 300 GB/s bi-directional bandwidth
PGI Compiler Option	-ta=tesla:cc70	-ta=tesla:cc70
	Inter-node Network	
IB Device on Node	2 EDR 100 Gb 2-port cards	2 EDR 100 Gb 2-port cards
	Local Disk	
SSD/node	** 3.2 TB raw, 1.6 TB usable (mirrored) (2 x 1.6 TB SAS SSD connected via an internal smart Raid card)	** 3.2 TB raw, 1.6 TB usable (mirrored) (2 x 1.6 TB SAS SSD connected via an internal smart Raid card)

* r101i0n[12-13] are for the two nodes containing eight V100 cards.

** Not yet available for public usage, pending configuration decisions.

For more hardware details, access a GPU node through a PBS session and run one of the following commands:

For CPU info:

```
cat /proc/cpuinfo
```

For host memory info:

```
cat /proc/meminfo
```

For GPU info:

```
/usr/bin/nvidia-smi -q
```

or load a PGI compiler module, for example, comp-pgi/20.4, and run the command pgaccelinfo.

```
module use -a /nasa/modulefiles/testing
module avail comp-pgi
---- /nasa/modulefiles/testing ----
comp-pgi/17.10 comp-pgi/18.10 comp-pgi/18.4 comp-pgi/19.10 comp-pgi/19.5 comp-pgi/20.4
---- /nasa/modulefiles/sles12 -----
comp-pgi/16.10 comp-pgi/17.1

module load comp-pgi/20.4
pgaccelinfo
```

Currently, no direct communication exists between a GPU on one node and a GPU on another node. If such communication is required, the data must go from the GPU to the CPU via the PCI Express bus, and from one CPU to another via InfiniBand network using MPI.

The two deep learning frameworks supported on NAS systems, TensorFlow and PyTorch, are capable of training using multiple GPUs on the same node. In TensorFlow, the MirroredStrategy function is the best option for training using multiple GPUs on a single node. In PyTorch, the DataParallel function is the best option.

Using TensorFlow MirroredStrategy

Using the TensorFlow MirroredStrategy framework is relatively straightforward, with only slight modifications needed to existing Python code. Complete these steps to set it up.

1. Create a MirroredStrategy instance:

   ```
   mirrored_strategy = tf.distribute.MirroredStrategy()
   ```

2. Create, compile, and fit the model in the scope of the MirroredStrategy:

   ```
   with mirrored_strategy.scope():
     model = Model()
     model.compile(loss = ..., optimizer = ..., ...)
     model.fit(x_train, y_train, batch_size, epochs, ...)
   ```

   This approach allows TensorFlow to effectively use all of the GPUs attached to the node. If you want to use only a few of the GPUs in a node to train a model, then you must assign the GPUs to the MirroredStrategy, as follows:

   ```
   mirrored_strategy = tf.distribute.MirroredStrategy(devices=["/gpu:0", "/gpu:1"])
   ```

   The model will be trained using only the assigned GPUs.

Using PyTorch DataParallel

The basic principle behind PyTorch DataParallel is to allocate the data and model to the GPUs, rather than the CPUs. Complete these steps to set it up.

1. Create the device object:

   ```
   device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
   ```

2. Move the model to the device:

   ```
   model = nn_model()
   model = nn.DataParallel(model)
   model.to(device)
   ```

3. Assign the data to the GPUs by applying the *.cuda() function to the torch tensor. For example:

   ```
   gpu_data = torch.tensor(data).cuda()
   ```

The model can now be trained on the GPUs using existing PyTorch code.

Related Resources

- Distributed training with TensorFlow
- Multi-worker training with Keras
- PyTorch Tutorial: Data Parallelism

Dask is a Pythonic distributed data science framework that enables development of scalable Python code for big data. You can think of Dask as a Python library for parallel computing, and also as a tool to scale computational libraries such as Numpy, Pandas, and Scikit-Learn.

To use Dask, load and activate one of our conda environments for machine learning that has the Dask software package installed.

This article provides a very basic overview of how Dask parallelizes common Python operations. For detailed information, see the Dask website.

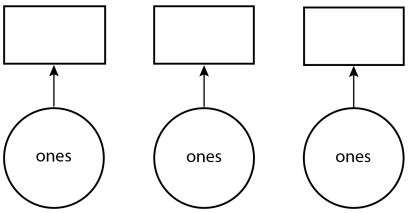
Parallelizing Python Operations

This section provides three different examples of parallelization in Dask.

Example 1

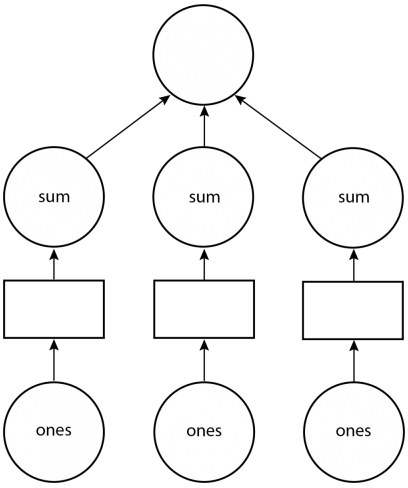
To create a one-dimensional array with entries equal to unity:

```
import dask.array as da
x = da.ones(15, chunks=(5,))
```



To sum the array:

```
x.sum
```

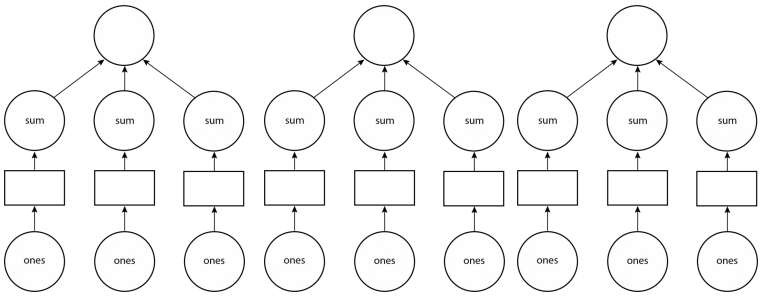


There are three groups, each with five chunks of data (3x5) to be summed in parallel.

Example 2

To sum a two-dimensional array:

```
import dask.array as da
x = da.ones((15,15), chunks=(5,5))
x.sum(axis=0)
```



The 3x3 groups are summed in parallel.

Example 3

To add an array to its transpose:

```
import dask.array as da
x = da.ones((15,15), chunks=(5,5))
x + x.T
```

This is done by obtaining the transposes in parallel before adding the original matrix.

Parallelizing Existing Systems

Dask can scale existing codebases with minor changes. For example, consider the following code:

```
results = []

for x in A:
  for y in B:
    if x < y:
      results.append(f(x,y))
    else :
      results.append(g(x,y))
```

Suppose there is no parallelization for the work at `f` and `g`, because it does not look like a big array or dataframe. You can use the dask.delayed function to parallelize the existing codebase:

```
import dask
results = []

for x in A:
  for y in B:
    if x < y:
      results.append(dask.delayed(f)(x,y))
    else :
      results.append(dask.delayed(g)(x,y))

Results = dask.compute(results)
```

Running Across Multiple Nodes

To run Dask across multiple nodes, load the `mpi-hpe/mpt` module and use the `dask_mpi conda` environment:

```
PBS r429i5n7~> module use /swbuild/analytix/tools/modulefiles
PBS r429i5n7~> module load miniconda3/v4
PBS r429i5n7~> module load mpi-hpe/mpt
PBS r429i5n7~> source activate dask_mpi
```

At the command prompt on the master node, run `dask-mpi`:

```
PBS r429i5n7~> mpirun -np 20 dask-mpi --worker-class distributed.Worker --scheduler-file
~/dask_sch/sched.json
```

where `~/dask_sch` is the path to the Dask scheduler file, `sched.json`.

Dask will now run across the nodes, and the scheduler information can be found in the`~/dask_sch/sched.json` directory.

You can use this sample PBS script:

```
##!/bin/bash
#PBS -q devel -l select=25:ncpus=4:model=has
#PBS -l walltime=01:00:00
#PBS -j oe
#PBS -N air_dask_test

#set environment
```

```
module use /swbuild/analytix/tools/modulefiles
module load miniconda3/v4
module load mpi-hpe/mpt
source activate dask_mpi

#clear and create directory for dask scheduler
if [ -d dask_sch ]
then
   echo "removing old scheduler directory"
   rm -r dask_sch
   echo "creating new scheduler directory"
   mkdir dask_sch
else
   echo "creating scheduler directory"
   mkdir dask_sch
fi

#run program
mpirun -np 20 dask-mpi --worker-class distributed.Worker --scheduler-file
$(pwd)/dask_sch/sched.json & your_python_script
```

The What-If Tool (WIT) is a helpful visualization tool that aids in analyzing models and datasets. The tool automatically visualizes datasets and allows you to compare different models and see how specific inputs change the output. WIT can visualize both classification and regression models, and gives various visualized statistics like receiver operating characteristic (ROC) curves, confusion matrices, and error calculations, depending on the type of model.

WIT is included in the latest versions of TensorFlow environments installed on Pleiades. To use WIT, you must first set up Jupyter Notebook for your NAS environment.

Note: WIT only works on Broadwell or newer nodes.

Starting a Jupyter Notebook

You can access WIT through an interactive Jupyter notebook. To use WIT, first start a Jupyter notebook by using the steps described in Using Jupyter Notebook for Machine Learning Development on NAS Systems—with one change in Step 4:

- In Step 4, instead of loading a Jupyter lab server, load a Jupyter notebook server, as WIT does not work in Jupyter Lab. The command line becomes:

  (*env_name*)r313i0n0 % jupyter notebook --no-browser

The other steps remain the same.

Configuring the Notebook to Use WIT

Once you load a Jupyter notebook in your local browser, you can configure it to use WIT. Complete these steps:

1. In a Python cell of a notebook, load the proper libraries:

   ```
   from witwidget.notebook.visualization import WitConfigBuilder
   from witwidget.notebook.visualization import WitWidget
   ```

2. Enable your dataset to work properly in the visualizer by converting your data to one of the following data structures:
   - tf.train.Example
   - tf.train.SequenceExample protos

   For more information on these data structures, see this documentation on the TensorFlow website.
3. Create the tool by building a WitConfigBuilder, which specifies the data and model to be used for the visualization. The simplest way to use any model is with a custom prediction function:

   ```
   config_builder = WitConfigBuilder(data).set_custom_predict_fn(model)
   ```

   If you want to compare two models, you can add:

   ```
   config_builder=
   WitConfigBuilder(data).set_custom_predict_fn(model).set_custom_compare_predict_fn(model2)
   ```

   where data is a list of tf.train.Example or tf.train.SequenceExample protos, and model is any function used for prediction. The tool directly calls this function for inference with the data provided.
4. Start an interactive window by creating a WitWidget using the following configuration:

   ```
   a = WitWidget(config_builder, height=tool_height_in_px)
   ```

   where config_builder is the WitConfigBuilder object created above, and height is the height (in pixels) of the interactive window opened in the notebook. We recommend setting the height at 1000, depending on the size of your browser window.

Completing these steps will display the interactive window in the notebook when you run the cell. The window will have three tabs at the top:

- **Tab 1:** visualizes all the data.
- **Tab 2:** visualizes the model results.
- **Tab 3:** visualizes data statistics by features.

Additional Resources

- For more information and in-depth examples, see the WIT website .
- To learn more about what you can do with WIT, review the notebooks provided by the developers.

You can improve performance or use resources more efficiently using the information below.

Environment Variables (CPU-only)

If you are running your machine learning environment on CPUs, you can set and adjust the environment variables KMP_AFFINITY, KMP_SETTINGS, OMP_NUM_THREADS, and KMP_BLOCKTIME to improve performance.

For example, we used the following settings for an MNIST test case, using TensorFlow 1.9 built with CUDA 9 on a 20-core Ivy Bridge node:

```
setenv KMP_AFFINITY "granularity=fine,verbose,compact,1,0"
setenv KMP_SETTINGS 1
setenv OMP_NUM_THREADS 20
setenv KMP_BLOCKTIME 30
```

In the test case, we achieved the following improvement:

Settings	Timing
Default	497.664 seconds
Adjusted	127.677 seconds

Note: The best performance may require further adjustments, depending on which CPUs you are using.

TensorFlow Performance and Efficiency

Setting the BATCH_SIZE Parameter

In TensorFlow, you can improve performance by increasing the value of the BATCH_SIZE parameter from the default, in order to increase the matrix size and improve the performance as high as the hardware can handle. However, keep in mind that increasing the value will degrade the accuracy of the model.

Loading CUDA Modules to Improve Efficiency

If you are getting PTX driver errors when running TensorFlow code on the GPU nodes, you can load the CUDA 10.1 module after you load the TensorFlow environment. In the PBS script, load the modules in this order:

```
module -a use /swbuild/analytix/tools/modulefiles
module load miniconda3/v4
source activate tf2_3
module load cuda/10.1
```

This should remove the errors and help reduce training times.

Using a Single GPU Card

Each NVIDIA V100 GPU node contains multiple GPU cards (either 4 or 8). If you think fewer cards are sufficient for your application, you can use the GPU nodes more efficiently by using the cards separately.

The PBS v100 queue allows jobs to occupy virtual nodes, called vnodes, that can be a small portion of a physical node. For example, suppose you want to use just one GPU for a non-MPI program, where the program uses two OpenMP threads on two cores and 40 GB of memory. You can specify this in your PBS script as follows:

```
#PBS -l select=1:model=sky_gpu:ngpus=1:ncpus=2:ompthreads=2:mem=40g
#PBS -l place=vscatter:shared
```

For more detailed information about using the v100 queue, see the following articles:

- Requesting GPU Resources
- PBS Job Requests for V100 GPU Resources

However, if you requested more GPU cards and want to use one or more of them (for example, for debugging purposes), you can use the following methods. For example, if you want only the 0th card to be visible to TensorFlow, but you want to use all memory in the GPU node, add the following lines to your python code:

```
import os
os.environ["CUDA_VISIBLE_DEVICES"]="0"
```

If you want to use more than one card""for example, the 0th and 2nd cards""you would replace "0" with "0,2".

If you want only the 0th card to be visible to TensorFlow, and only allocate the memory associated with the 0th card, also add these lines to your python code:

```
config = tf.ConfigProto()
config.gpu_options.allow_growth=True
sess = tf.Session(config=config)
tf.keras.backend.set_session(sess)

def _get_available_devices():
   from tensorflow.python.client import device_lib
   local_device_protos = device_lib.list_local_devices()
   return [x.name for x in local_device_protos]

print _get_available_devices()
```

The function get_available_devices() indicates the value that you set for CUDA_VISIBLE_DEVICES. For example, if you set os.environ["CUDA_VISIBLE_DEVICES"]="0", the function would print the following line:

```
u'/device:CPU:0', u'/device:GPU:0'
```

This indicates that the GPU device visible to CUDA is "0" under the CPU ID "0".

Machine Learning FAQs

What data analytics tools does NAS provide for machine learning?

For a quick glance at the software that is currently provided in our machine learning environments, see Machine Learning: Overview.

You can also complete these steps to see the installed environments:

1. Load the miniconda3 module:

   ```
   % module use -a /swbuild/analytix/tools/modulefiles
   % module load miniconda3/v4
   ```

2. Run the following command to see the available environments:

   ```
   % conda info --envs
   ```

Is Jupyter Notebook available?

Jupyter Notebook, an open-source application based on Interactive Python, is a useful tool for interactively exploring science data. You cannot run Jupyter on a Pleiades front-end (PFE) system; rather, you must run it on a compute note dedicated to you alone.

To learn more about using Jupyter Notebook on NAS systems, see Using Jupyter Notebook for Machine Learning Development.

Note: A one-time security setup is required before you can begin using Jupyter Notebook.

Which environments can I use to complete Step 2 in the Secure Setup for Jupyter Notebooks?

All of our provided environments have Jupyter installed, so you can choose any of them, besides the base environment. You can find the available environments by using the `conda info -envs` command. We suggest using the latest TensorFlow environment (currently, `tf2_4`).

Do I need to install Jupyter myself?

You do not need to install Jupyter for yourself in order to do thesecure setup step, or use it afterwards. Loading any provided environment will work in the secure setup step, and the completed procedure will allow you to use any of the other environments with Juypter installed.

Can I run a Jupyter Notebook without using the web-based user interface?

You cannot directly run a Jupyter Notebook without using the web-based user interface. However, you can convert your notebook into a Python script and run the script though the regular Python shell command.

To convert a notebook into a Python script:

```
% jupyter nbconvert --to script notebook.ipynb
```

This will create a *.py* file of the specified notebook, with each cell run in sequential order.

How do I run multiple instances of Python on the compute nodes?

To learn how to run multiple instances of a Python code on the compute nodes, with each instance running your code, seeUsing GNU Parallel to Package Multiple Jobs (Instances) in a Single PBS Job.

How do I use multiple nodes with TensorFlow?

To use multiple nodes, you can use TensorFlow's "MultiWorkerMirroredStrategy" variable, which is described in their article Distributed Training with TensorFlow.

To call the strategy in your code, use:

```
if dist_nodes :
    strategy=tf.distribute.experimental.MultiWorkerMirroredStrategy (
          tf.distribute.experimental.CollectiveCommunication.AUTO )
  else :
    strategy = tf.distribute.MirroredStrategy()
```

The collective operations include the following options:

CollectiveCommunication.RING
: Implements ring-based collectives using gRPC as the communication layer.

CollectiveCommunication.NCCL
: Uses NVIDIA's NCCL to implement collectives.

CollectiveCommunication.AUTO
: Defers the choice to the runtime.

You also need to set up the TF_CONFIG environment variable, which is required by TensorFlow:

```
if dist_nodes :
   index = int(sys.argv[1])  # $@ has n indices indicating n nodes
   verbose = index < 1       # verbose is True only when index = 0
   nodes=[]
   # node names for each node, append a port # to the names
   for i in range(2,len(sys.argv)):
     nodes.append(sys.argv[i] + ':2001')
   # set TF_CONFIG variable that MultiWorkerMirroredStrategy needs
   os.environ['TF_CONFIG'] = json.dumps({
      'cluster': {
         'worker': nodes
      },
      'task': {'type': 'worker', 'index': index}
   })
   print(os.environ['TF_CONFIG'])
 ##
```

For more details about TensorFlow programming and an example of how to use the above methodology, please see Multiple CPU Nodes and Training in TensorFlow.

How do I run parallel jobs using multiple GPU nodes?

The following steps show an example of how to run parallel jobs across NVIDIA Volta V100 GPU nodes. Adapt these steps to suit your needs.

1. Request the GPU nodes in your PBS script.

   To request V100 GPU nodes using the v100 queue:

   ```
   #!/bin/bash
   #PBS -l select=3:model=sky_gpu:mpiprocs=1:ncpus=36:ngpus=4:mem=300g
   #PBS -l place=scatter:excl
   #PBS -q v100
   #PBS -l walltime=1:00:00
   #PBS -j oe
   #PBS -N multinodetest
   ```

   The number of chunks (3) and the model type (sky_gpu) are specified in the select statement. Note that in order to get access to all the cores, GPUs, and memory on the V100 nodes, you must also specify these resources explicitly. In addition, in the place statement, you must specify exclusive access (excl).

   For more detailed information about using the v100 queue, see Requesting GPU Resources.

2. Use $PBS_NODEFILE to get node info:

   ```
   NODES=($( cat $PBS_NODEFILE | sort | uniq ))
   NUM_OF_NODES=${#NODES[@]}
   C=0
   ```

3. For each node, run *MyScript*.sh, which will invoke your Python code:

   ```
   # for each node that's not the current node
   for node in ${NODES[@]}
   do
     if [[ $node != $(eval hostname) ]]
     then
         ssh $node "$MYPATH/MYSCRIPT.sh $C ${NODES[*]}" &
         C=$(($C+1))
   # sleep 2 to start the worker nodes before the main node
         sleep 2
     fi
   done
   ```

   An example *MyScript*.sh is shown below.

MyScript.sh is the script that will invoke your Python code (main.py). For example:

```
#!/bin/bash
#load module command
source /usr/local/lib/global.profile
cd /nobackup/$USER/vgg
BASE=/nobackup/$USER/
echo "On node: "
hostname
#
module purge
module -a use /swbuild/analytix/tools/modulefiles
module load miniconda3/v4
source activate env_name
...
let PARAMETER=
python $BASE/main.py $PARAMETER  $@
```